TECHNICAL REPORT 94-09

The Transformation Calculus

Jacques Garrigue

April 25, 1994

Department of Information Science
Faculty of Science, University of Tokyo

7–3–1 Hongo, Bunkyo-Ku Tokyo, 113 Japan

TITLE
The Transformation Calculus

AUTHORS
Jacques Garrigue

KEY WORDS AND PHRASES
λ-calculus, algorithms, functional programming, imperative programming

ABSTRACT
The lambda-calculus, by its ability to express any computable function, is theoretically able to represent any algorithm. However, notwithstanding their equivalence in expressiveness, it is not so easy to find a natural translation for algorithms described in an imperative way.
The transformation calculus, a conservative extension of lambda-calculus, corrects this flaw by re-introducing an implicit notion of state in the calculus.
This calculus stays very close to lambda-calculus, and keeps most of its properties. We prove here its confluence.

REPORT DATE	**WRITTEN LANGUAGE**
April 25, 1994	English
TOTAL NO. OF PAGES	**NO. OF REFERENCES**
19	17

ANY OTHER IDENTIFYING INFORMATION OF THIS REPORT

DISTRIBUTION STATEMENT
First issue 40 copies. This technical report is available via anonymous FTP from `ftp.is.s.u-tokyo.ac.jp` (directory `/pub/tech-reports`).

SUPPLEMENTARY NOTES

DEPARTMENT OF INFORMATION SCIENCE
Faculty of Science, University of Tokyo
7-3-1 Hongo, Bunkyo-ku Tokyo, 113 Japan

The Transformation Calculus

Jacques Garrigue

The University of Tokyo, Faculty of Science
Dept. of Information Science, Yonezawa Lab
Hongo 7-3-1, Bunkyo-ku, TOKYO 113.
garrigue@is.s.u-tokyo.ac.jp

May 3, 1994

Abstract

The lambda-calculus, by its ability to express any computable function, is theoretically able to represent any algorithm. However, notwithstanding their equivalence in expressiveness, it is not so easy to find a natural translation for algorithms described in an imperative way.

The transformation calculus, a conservative extension of lambda-calculus, corrects this flaw by re-introducing an implicit notion of state in the calculus.

This calculus stays very close to lambda-calculus, and keeps most of its properties. We prove here its confluence.

1 Introduction

Whereas the lambda-calculus is theoretically able to represent any sequential algorithm, encoding from an imperative style, using mutable variables, is by no way immediate. What we will propose here is a generalization of lambda-calculus in which such algorithms find a natural expression.

The need for such an algorithmic extension of lambda-calculus appears from two sides. First, it would be interesting to be able to write directly algorithms in the lambda-calculus, to make more evident the equivalence of both systems. Ideally, a syntactic sugar would be perfect, but the problem is too complex for that. So, we will need a non-trivial extension, in which terms may be given an imperative meaning, but are still close enough to lambda-terms.

Second, there is a demand from the functional programming community. The problem looks more general than having mutable variables, since it includes input-outputs and side-effects in general. But it is shown that all these topics can be seen in terms of modifications of a global state, which translates easily into mutable variables.

The author is only aware of one partial answer to the first concern up to now. In [14], Odersky, Rabin and Hudak propose an extension of lambda-calculus with operations on a store. Stores are mapping from names to values, which represent a mutable state, and have to be modified according to sequential semantics. Their system, by using explicit substitutions, somehow like in the $\lambda\sigma$-calculus [1], for operations on the store, keeps quite close to a lambda-calculus where reduction order is free. However, the extension is orthogonal, in that new constructs are introduced for all operations on the store. This

means redundancies, for instance two different abstractions for reading an argument and reading from the store. That seems not enough to express a relation between imperative algorithms and the lambda-calculus.

The second point has been paid more attention lastly. In addition to the above work, appliable on this side, which needs only a weaker integration of the functional and algorithmic part, we may cite Lamping's *Unified parameterization system* [11] where one can defines functions transforming the store, and use them in a special let construct. Only an operational semantics is defined. Recent works on monads, initiated by Moggi, deal with this problem too, from semantic and type theoretic points of view [13, 17, 15], as well as Gifford's effect type systems [8, 16]. Continuations again are a way to tackle it, and the concept of partial continuation developed by Johnson and Duggan in [10] shows similarities with transformations as we will define them.

We remark here that all the works cited above are based on a store model, and mostly try to give it a formalization. Transformation calculus is not based on such a model.

The intuition of the transformation calculus is rather based on two other threads of work. The first one is the Categorical Combinatory Logic [5, 9], in which composition plays a central role. In some way we could have presented it under the first goal, as a means to write algorithms in something close to the lambda-calculus. However, if CCL is the basis for an environment-based abstract machine, the CAM [4], it seems difficult to directly write algorithms in it, since all operations are done through numerical indexes only. As a matter of fact, a subsystem of the transformation calculus, in which we do not use names, has a lot in common with CCL.

The second one is process calculi. Their use of names for communication is similar to the principle of the transformation calculus. In [3], Boudol proposes the γ-calculus. The base is lambda-calculus, but applications express emissions of messages and abstractions their reception, while multiple terms can be evaluated simultaneously. Milner's π-calculus [12] proceeds alike, and by labeling with names applications and abstractions, it allows the use of multiple channels. The fundamental difference with our calculus is that non-determinism of the receptor of a message make these calculi divergent, while our terms are sequenced in order to keep determinism.

The first idea of the transformation calculus, contrary to an orthogonal extension, is that there is no difference between reading a variable from an environment and reading an argument through an abstraction. Similarly writing it is similar to an application. In that we follow Lamping's idea, or process calculi if we see a variable as a communication link between its definition and its use. However our difference is in the labeling system we use to express where we are reading from and where we are writing to. Rather than a simple set of names, in which writing two times on the same name would result in destroying the previous value assigned to this name, or non-determinism between the two values, we developed a system using names indexed by integers. This system first appeared in the selective λ-calculus [2, 7]. In that version it proved to keep most properties of lambda-calculus, including confluence, typings or strong normalization of typable terms.

Transformation calculus extends selective λ-calculus by introducing composition between terms. The idea is to add a special constructor which, applied to any number of arguments, sends them back as a record. Compositions results in an application on this record. Functions ending with such a constructor, or *transformations*, may modify a representation of the state by handing back a record of transformed arguments. The

next term in the composition will receive these arguments just as if it had been directly applied to them. By this very simple mechanism, we can handle in an unified way arguments and mutable variables, and easily encode algorithms. Since compositions can be reduced as soon as a transformation is recognized as such (by the presence of the transformation constructor), terms tend to be normalized towards lambda-calculus (or rather selective λ-calculus) terms.

To make even more evident the relation between transformations and variable modifications we define the notion of *scope-free variable*, which can be encoded in the transformation calculus. Scope-free variables can simulate both global and local variables, but the zone of terms they affect can even be defined on a finer grain basis, from their creation to their destruction. Outside this zone, they have absolutely no effect, like local variables.

The rest of this abstract is composed as follows. In Section 2 we introduce progressively the different features which form the transformation calculus. In this process we are brought to define *streams*, which are formalized in Section 3. Section 4 is devoted to the formal definition of the transformation calculus. Section 5 defines, and gives the fundamental property of scope-free variables. Finally, Section 6 concludes. Proofs are given in appendix.

2 Composition and streams

We first introduce informally and progressively the features of our calculus. We start from the classical pure lambda-calculus, that is

$$M ::= x \mid \lambda x.M \mid MM$$

with β-reduction

$$(\lambda x.M)N \rightarrow_\beta M[x\backslash N]$$

and where terms are considered modulo α-conversion (renaming of bound variables).

2.1 Implicit currying

Currying is the fundamental transformation by which multi-argument functions are encoded in the lambda-calculus. It can appear in abstractions as well as applications. For instance $f(a,b)$ will be encoded as $(f(a))(b)$ ($f\,a\,b$ without parentheses), and $\lambda(x,y).M$ becomes $\lambda x.\lambda y.M$.

This operation does not modify the nature of calculations, since clearly $(\lambda(x,y).M)(a,b)$ and $(\lambda x.\lambda y.M)\,a\,b$ reduce to the same $M[x\backslash a, y\backslash b]$ (provided x and y are distinct variables). Currying can be extended to an arbitrary number of arguments, that is encoding $\lambda(x_1,\ldots,x_n).M$ as $\lambda x_1.\cdots.\lambda x_n.M$. As long as we encode similarly applications and abstractions, no problem should appear.

By implicit currying, we mean that we will write curried and uncurried versions of terms indifferently, always supposing that we reduce curried ones. Of course we work in the pure lambda-calculus without pairing, so that no confusion is possible. The new syntax becomes

$$M ::= x \mid \lambda(x,\ldots).M \mid M(M,\ldots)$$

where abstracted variables under the same λ should be distinct. Implicit currying is expressed by the two structural equivalences:

$$\begin{array}{rcll} \lambda(x_1,\ldots,x_n).M & \equiv & \lambda(x_1,\ldots,x_k).\lambda(x_{k+1},\ldots,x_n).M & \text{all } x_i\text{'s are distinct} \\ M(N_1,\ldots,N_n) & \equiv & M(N_1,\ldots,N_k)(N_{k+1},\ldots,N_n) & \end{array}$$

β-reduction applying on the implicitly curried form, a reduction step only binds one variable.

$$(\lambda(x,y).M)(a,b) \to_\beta (\lambda(y).M[x\backslash a])(b) \to_\beta M[x\backslash a, y\backslash b]$$

In fact, if we remember the habit many have of writing $(\lambda x\,y.M)\,a\,b$ for the above term, we have done absolutely nothing new. However this syntax lets us emphasize some natural groupings of values. For instance the encoding of pairs in lambda-calculus can be written as $\lambda(x,y).\lambda f.f(x,y)$.

2.2 Composition

The next step is to introduce a binary *composition* operator ("$\circ$") and a *transformation* constructor ("$\{\}$"). A transformation is what should appear on the right side of a composition, for a reduction to progress correctly.

$$M ::= \ldots \mid \{\} \mid M \circ M$$

Together we add two new rules, to reduce compositions. We just give here a weak version of the elimination rule ($\circ$), working only when transformations reduced to tuples, to avoid problems of variable bindings. The real calculus will use a strong one.

$$\begin{array}{lcl} M \circ (\{\}(N_1,\ldots,N_k)) & \to_\circ & M(N_1,\ldots,N_k) \\ (M \circ N)(P_1,\ldots,P_k) & \to_. & M \circ (N(P_1,\ldots,P_k)) \end{array}$$

We can see the sequencing role of composed pairs as follows: when we apply $(M \circ N)$ to a sufficient input tuple of arguments, we first apply N to this tuple, get (hopefully) a *tuple-term* (term of form $\{\}(N_1,\ldots,N_k)$) as result of its reduction, and apply M to this result tuple.

It just looks like we added a stack machine into the lambda-calculus. For instance, we can write the transformation that switches two terms on top of a stack as $sw = \lambda x.\lambda y.\{\}\,y\,x$, and can apply it to an input tuple of any size:

$$\begin{array}{lll} & (\lambda x.\lambda y.\{\}\,y\,x) & (a,b,c) \\ \to_\beta & (\lambda y.\{\}\,y\,a) & (b,c) \\ \to_\beta & (\{\}\,b\,a) & (c) \\ \equiv & \{\} & (b,a,c) \end{array}$$

Composed with another term, it plays the same role as the $C = \lambda f.\lambda(x,y).f(y,x)$ combinator; but in a postfix way.

$$\begin{array}{lll} & (K \circ sw)(a,b) & (c) \\ \equiv & (K \circ sw) & (a,b,c) \\ \to_. & sw & (a,b,c) \\ \overset{*}{\to} & \{\} & (b,a,c) \\ \to_\circ & (\lambda x.\lambda y.x) & (b,a,c) \\ \overset{*}{\to}_\beta & b & (c) \end{array}$$

Since we are in the lambda-calculus, we can define the fix-point operator Y. We just define then loops in terms of this operator. The functional for a while-do loop can be defined as

$$\text{while} = Y(\lambda whl.\lambda(end, do).((\lambda b.\text{if } b \text{ then } whl(end, do) \circ do \text{ else } \{\}) \circ end))$$

The *end*-condition is a transformation that adds to its input a boolean b, false to end, true to go on, leaving the rest in position. *do* may change the values from the input, but not their number. Such a functional works on a state of any size.

An imperative version of Euclid's algorithm for the greatest common divisor can then be written

$$(\lambda(x, y).y) \circ \text{while}(\lambda x.\{\}(x \neq 0, x), \lambda(x, y).\{\}(y \bmod x, x))$$

We notice here an important difference between this "while" functional and something equivalent written using pairing. Here our *end*-condition only uses x, whereas a functional using pairing would have required it to receive a pair eventhough y is not needed, for the sake of generality. This remark will become even more important when we will add to our calculus the power of *selective currying*.

2.3 Selective currying

Combining lambda-calculus and a stack machine should be enough to express algorithms both in their functional and imperative form. However, in chosing a reduction system rather than an equational theory to express our calculus, we are interested in giving some meaning to the reductions themselves. We can think of various meanings, like complexity, sequentiality constraints, *etc...* If we do such an analysis, then we see that,with simple implicit currying, we need much more reduction steps than should be necessary. That is, when we want to access the 5^{th} element of a tuple we have to extract successively all the elements before it, and put them back:

$$\lambda(x_1, x_2, x_3, x_4, x_5).\{\}(x_5, x_1, x_2, x_3, x_4)$$

Even more than the number of reductions, we should be preoccupied by the fact we have accessed four unrelated values to move only one. And this in an asymmetrical way, values after the fifth element being left untouched.

2.3.1 Indexed streams

The answer to this problem comes from the canonical injection of tuples into records, as it can be found functional languages like Standard ML or LIFE. That is

$$(x_1, \ldots, x_n) \equiv \{1 \Rightarrow x_1, \ldots, n \Rightarrow x_n\}$$

We will just consider tuples as a particular case of *streams* (the name we give to these numerically indexed records). We use streams to access directly the arguments we are interested in. For instance we write the previous transformation

$$\lambda\{5 \Rightarrow x\}.\{\}\{1 \Rightarrow x\}$$

We extract the 5^{th} element from the input, and put it in first position.

By symmetry we can use such incomplete streams in applications as well as abstractions. The transformation $\{\}\{5 \Rightarrow a\}$ inserts a before the fifth argument of its input, pushing up all its followers by one.

$$\{\}\{5 \Rightarrow a\}(b_1, b_2, b_3, b_4, b_5, b_6) \rightarrow \{\}(b_1, b_2, b_3, b_4, a, b_5, b_6)$$

The stream we apply our transformation to can be incomplete too, like in the following case.

$$\{\}\{5 \Rightarrow a\}\{2 \Rightarrow b, 5 \Rightarrow c, 7 \Rightarrow d\} \rightarrow \{\}\{2 \Rightarrow b, 5 \Rightarrow a, 6 \Rightarrow b, 8 \Rightarrow d\}$$

Generally, we must define a concatenation operation on streams, compatible with the partial isomorphism between streams and tuple. This is done by shifting indexes in the second stream according to those present in the first one. The algorithm doing that will be detailed in the next section. The important point is that we have a reciprocal operation, sub-stream extraction, which we can use to separate a stream into two parts, forming it back by concatenation.

$$\begin{array}{ll} & (\lambda\{2 \Rightarrow x, 3 \Rightarrow y\}.M)\{1 \Rightarrow a, 2 \Rightarrow b, 3 \Rightarrow c, 5 \Rightarrow d, 7 \Rightarrow e\} \\ \equiv & (\lambda\{2 \Rightarrow x, 3 \Rightarrow y\}.M)\{2 \Rightarrow b, 3 \Rightarrow c\}\{1 \Rightarrow a, 3 \Rightarrow d, 5 \Rightarrow e\} \\ \xrightarrow{2}_{\beta} & M[x \backslash b, y \backslash c]\{1 \Rightarrow a, 3 \Rightarrow d, 5 \Rightarrow e\} \end{array}$$

This operation can be applied to the abstraction part too:

$$\begin{array}{ll} & (\lambda\{2 \Rightarrow x, 3 \Rightarrow y\}.M)\{2 \Rightarrow a, 4 \Rightarrow b\} \\ \equiv & (\lambda\{2 \Rightarrow x\}.\lambda\{2 \Rightarrow y\}.M)\{2 \Rightarrow a\}\{3 \Rightarrow b\} \\ \rightarrow_{\beta} & (\lambda\{2 \Rightarrow y\}.M[x \backslash a])\{3 \Rightarrow b\} \\ \rightarrow_{..} & \lambda\{2 \Rightarrow y\}.(M[x \backslash a]\{2 \Rightarrow b\}) \end{array}$$

In the second line we decompose both abstraction and application to permit β-reduction in the third one. The $\rightarrow_{..}$ rule in the last line is there to switch unrelated abstractions and applications, and let the reduction progress smoothly. We will argue later its coherence with the rest of the system.

Thanks to these streams we can now write algorithms using the power of something quite close to a direct-access stack machine. It is slightly different since reading a position in the stream destroys this position, but writing resulting in a symmetrical insertion this is not a problem. Here is a transformation incrementing the fifth position in a stream.

$$\begin{array}{lll} & \lambda\{5 \Rightarrow x\}.\{\}\{5 \Rightarrow x + 1\} & (a, b, c, d, e) \\ \equiv & (\lambda\{5 \Rightarrow x\}.\{\}\{5 \Rightarrow x + 1\})\{5 \Rightarrow e\} & (a, b, c, d) \\ \rightarrow_{\beta} & \{\}\{5 \Rightarrow e + 1\} & (a, b, c, d) \\ \equiv & \{\} & (a, b, c, d, e + 1) \end{array}$$

2.3.2 Naming positions

The problem of such a system is that since the indexes in the stream may change with each transformation we apply to it, we have no uniform way to address a defined position in it. This increments the fifth position by the first (which is destroyed):

$$\lambda\{1 \Rightarrow x, 5 \Rightarrow y\}.\{\}\{4 \Rightarrow x + y\}$$

The position we addressed as 5^{th} before the transformation must become the 4^{th} after it, since we do not distinguish arguments (x) from mutable variables (y).

This possibility of mixing is good, since it means that we can see everything with a functional insight. However we would like to have a more uniform way to handle a position. Going on with our analogy between streams and records, we will accept to have named fields in our streams. So that we can write the previous incrementer as

$$\lambda\{1 \Rightarrow x, i \Rightarrow y\}.\{\}\{i \Rightarrow x + y\}$$

Since i is a named position its index is not modified by the extraction of the first one.

A problem may appear when we concatenate two streams containing the same named position. On records this operation has two definitions. Either we just refuse to do it (symmetrical concatenation), either we accept, take for this field the value in the right operand, and just forget the value in the left one (asymmetrical concatenation). Since concatenation is already asymmetrical on numerical indexes, there is no point in refusing to do such a thing. However we cannot erase a value since this would make our calculus diverge ($(\lambda\{i \Rightarrow x\}.\lambda\{i \Rightarrow y\}.M)\{i \Rightarrow a\}\{i \Rightarrow b\} \xrightarrow{2}_{\beta} M[x\backslash a, y\backslash b] \not\Downarrow (\lambda\{i \Rightarrow x\}.\lambda\{i \Rightarrow y\}.M)\{i \Rightarrow b\}$). So we just add a numerical index to our position name. That is, we view both numerical and name positions as their injections into their product, the set of labels. We have a default name ϵ such that the index n is in fact ϵn, and we add 1 to names so that p is $p1$. With that we define easily

$$M\{p \Rightarrow a\}\{p \Rightarrow b\} \equiv M\{p1 \Rightarrow a, p2 \Rightarrow b\}$$

This works right with the above example, which becomes $(\lambda\{i1 \Rightarrow x, i2 \Rightarrow y\}.M)\{i1 \Rightarrow a, i2 \Rightarrow b\}$.

Going on with our comparison with stacks, we have now as many stacks as we have names, each of them handled through indexed extraction and insertion.

We see here a new version of Euclid's algorithm, using labels for both the function and the while functional.

$$\begin{aligned} gcd = \quad & (\lambda\{m \Rightarrow m, n \Rightarrow n\}.n) \\ & \circ \text{while}\{end \Rightarrow \lambda\{m \Rightarrow m\}.\{ok \Rightarrow m \neq 0, m \Rightarrow m\}, \\ & \qquad do \Rightarrow \lambda\{m \Rightarrow m, n \Rightarrow n\}.\{m \Rightarrow n \bmod m, n \Rightarrow m\}\} \end{aligned}$$

(We will abbreviate $\{\}\{p \Rightarrow M, \ldots\}$ into $\{p \Rightarrow M, \ldots\}$ when there is no possible confusion, for the sake of readability.)

On such an example the addition of labels may look as pure verbosity, but what we obtain here is very close to what we would write as an imperative algorithm. We have only to add trivial abstractions of the form $\lambda\{m \Rightarrow m\}$ in order to transform an assignment-like syntax into functions.

2.4 Scope-free variables

Up to this point we have progressively enriched the lambda-calculus with new constructs. The transformation calculus is approximately the result of this process. We have insisted on how this calculus was a potential basis for an integration of imperative and functional styles in the design of algorithms. Here we introduce a general method to directly map the imperative notion of variable into the transformation calculus.

In fact, what we mean by *scope-free variable* is slightly stronger than a mutable variable. We call it scope-free, since it is not syntactically scoped like in structured

programming, neither is it global. We can say that it is local to a sequence of transformations, composed together.

A scope free variable is essentially a name v whose use in labels is exclusively reserved in the concerned sequence of transformations. This sequence is delimited by the creation of the variable with value a, encoded $\{\}\{v \Rightarrow a\}$, and its destruction by an abstraction, $\lambda\{v \Rightarrow x\}.\{\}$. Between these, all transformations using or modifying this variable should once take it (through abstraction) and then put it back (by application), identical or modified. Typically a modification can be written $\lambda\{v \Rightarrow x\}.\{v \Rightarrow M\}$. That is, the sequence has form:

$$(\lambda\{v \Rightarrow x\}.M) \circ \cdots \circ (\lambda\{v \Rightarrow x\}.\{v \Rightarrow M\}) \circ \ldots \circ (\{\}\{v \Rightarrow a\})$$

Since some transformations may be functionals, the recognition of such a structure is not immediate, but for instance m and n in the last version of Euclid's algorithm, where we mask labels (i.e. $\lambda(x,y).gcd\{m \Rightarrow x, n \Rightarrow y\}$), are scope-free variables.

The most interesting property of scope-free variables is that, similarly to scoped variables, they have no effect outside of the sequence they are used in. That is, we can use the same label v outside of the sequence our scope-free variable is local to, without interference. A scope-free variable may even be used in a subsequence of another scope-free variable using the same label:

$$(\lambda\{v \Rightarrow x\}.\{\}) \circ \cdots \circ (\lambda\{v \Rightarrow x\}.\{\}) \circ \cdots \circ \{v \Rightarrow b\} \circ \cdots \circ \{v \Rightarrow a\}$$

In the subsequence the external scope-free variable is identifiable by the label $v+1$ but comes back to v after.

Still, we must be careful that scope-free variables are not variables in the meaning of lambda-calculus: they appear on a completely different level, that of labels. Nor are they pervasive like would be references. We do not add side-effects to functions, but just provide some implicit way to manipulate a "stream" of arguments. That means that a function that is not called directly on this stream (through composition) will not access the scope-free variables it contains, and as such cannot have any imperative behaviour with respect to this stream. This is this limitation which permits us to assimilate scope-free variables with arguments, and still be a conservative extension of lambda-calculus.

3 Stream monoid

The transformation calculus is essentially defined in terms of operations on streams. We will first study them, and particularly their concatenation, which, being associative, makes their set a monoid. We will note the concatenation on streams by a simple dot "$\cdot$", and the monoid of streams is $(\mathcal{S}, \cdot)$.

Definition 1 *The set $\mathcal{S}(\mathcal{L}, \mathcal{A})$ of streams on a domain $\mathcal{A}$ is the set of finite partial functions from $\mathcal{L} = \mathcal{L}_s \times \mathcal{N}$, the set of labels, to $\mathcal{A}$. $\mathcal{L}_s$ is an ordered set of names, $\mathcal{N}$ natural integers without 0, and $\mathcal{L}$ is lexicographically ordered. A sub-stream is a subset of a stream.*

$$\mathcal{S} = \{s \in \mathcal{P}(\mathcal{L} \times \mathcal{A}) \mid s \text{ finite and } \forall (l,a),(l',a') \in s,\ l = l' \Rightarrow a = a'\}$$

Notations

- l will always represent an element of $\mathcal{L}$; p, q elements of $\mathcal{L}_s$; m, n elements of $\mathcal{N}$.
- $\mathcal{D}_s$ is the definition domain (first projection) of a stream s.
- $\{\}$ is the function defined nowhere ($\mathcal{D}_{\{\}} = \emptyset$).
- We note labels pn, and defining pairs $\{pn \Rightarrow a\}$ with $a \in \mathcal{A}$.
- The following equivalences of notation are admitted for labels and streams:
  - n denotes ϵn, p denotes $p1$
  - if $l = pn$ then $l + m = p(n + m)$
  - $(a_1, \ldots, a_n) = \{1 \Rightarrow a_1, \ldots, n \Rightarrow a_n\}$

Concatenation in $\mathcal{S}$ has to be coherent with its sub-monoids. The simplest is that of tuples: $\mathcal{L}_s = \{\epsilon\}$ (a singleton) and $\mathcal{S} = \bigcup_{n \geq 0} \mathcal{A}^{[\![1,n]\!]}$. Then concatenation is $(a_1, \ldots, a_m) \cdot (b_1, \ldots, b_n) = (a_1, \ldots, a_m, b_1, \ldots, b_n)$. This gives our first calculus, based on implicit currying alone.

A slightly stronger one is the same, with a bigger $\mathcal{L}_s$. We have just to restrict $\mathcal{D}_s$ to be of the form $\bigcup_{p \in \mathcal{L}_s} \{pi\}_{i=1}^{n_s(p)}$ for each stream ($n_s : \mathcal{L}_s \to (\mathcal{N} \cup \{0\})$ characterizes $\mathcal{D}_s$), and concatenation is done as above, but independently for each name. We have then a commutation monoid where the condition for commuting is having a different name. This is the basis for a calculus without partial streams.

In the general case, where we do not put any condition on the way streams are partial, we need the following, slightly more complex definition, permitting index operations.

Definition 2 (free position) *1. The n^{th} position on p in a stream r is said to be* occupied *if r is such that $pn \in \mathcal{D}_r$.*

2. $o_r(pn)$ is the offset *of n on p in r, that is the number of occupied positions in r on name p and whose index is less than or equal to n. Namely, $o_r(pn) = |\{pi\}_{i=1}^{n} \cap \mathcal{D}_r|$.*

3. The n^{th} free position *for p in r is given by $\phi_{r,p}(n) = \min\{i \in \mathcal{N} \mid i - o_r(pi) = n\}$. This function extends to streams by $\phi_r(\{p_i n_i \Rightarrow a_i\}_{i=1}^{k}) = \{p_i \phi_{r,p_i}(n_i) \Rightarrow a_i\}_{i=1}^{k}$.*

For instance, in $\{p1 \Rightarrow a, p3 \Rightarrow b, p5 \Rightarrow c, q2 \Rightarrow d\}$, the offset of $p3$ is 2, and that of $p4$ too. The offset of $q1$ is 0, and that of $qn, n > 1$ is 1. As a result, the second free position on p is 4 ($4 - o(p4) = 2$), and on q this is 3 ($3 - o(p3) = 2$). The formula for free positions works because $o_r(pn) - n$ is constant on occupied positions of r (both offset and index grow) and increases by 1 on free ones.

Definition 3 (concatenation) *Stream concatenation is defined as $r \cdot s = r \uplus \phi_r(s)$ where "$\uplus$" denotes union of (set represented) functions on disjoint domains.*

The intuition behind this definition is that when we do $r \cdot s$ we insert elements of s at free positions in r: for each name p we insert the element whose index is n in s at the n^{th} free position for p in r. ϕ_r is the function which does this shifting.

The next proposition states that $\mathcal{S}$ is a monoid, and $\{\}$ its neutral element.

Proposition 1 *Concatenation as in definition 2 is an associative application $\mathcal{S} \times \mathcal{S} \to \mathcal{S}$, accepting $\{\}$ as neutral element.*

PROOF Associativity comes from the equality $\phi_r \circ \phi_s = \phi_{r \uplus \phi_r(s)}$. For this we reason on inverses: $\phi^{-1}_{s,p}(\phi^{-1}_{r,p}(i)) = i - d_r(pi) - d_s(\phi^{-1}_{r,p}(i)) = i - d_{r \uplus \phi_r(s)}(pi) = \phi^{-1}_{r \uplus \phi_r(s),p}(i)$. We then have $r \cdot (s \cdot t) = r \uplus \phi_r(s \uplus \phi_s(t)) = r \uplus \phi_r(s) \uplus \phi_r(\phi_s(t)) = r \uplus \phi_r(s) \uplus \phi_{r \uplus \phi_r(s)}(t) = (r \cdot s) \cdot t$.

$r \cdot \{\} = r = \{\} \cdot r$ is immediate ($\phi_{\{\}} = id$). □

We insisted in the preceding section on the fact we could build the transformation calculus if we had proper concatenation and sub-stream extraction on streams. We have concatenation, to get extraction we must be able to reverse it. For that we just need an inverse to ϕ_r. The next proposition gives it to us.

Proposition 2 *ϕ_r is a bijection from $\mathcal{S}$ to $\{s \in \mathcal{S} \mid \mathcal{D}_r \cap \mathcal{D}_s = \emptyset\}$.*

PROOF For each name p we chose $\phi^{-1}_{r,p}(n) = n - o_r(pn)$. For any label pn, $\phi^{-1}_{r,p}(\phi_{r,p}(n)) = \phi^{-1}_{r,p}(\min\{i \mid i - o_r(pi) = n\}) = n$. Moreover, $\phi_{r,p}(\mathcal{N}) = \{n \in \mathcal{N} \mid pn \notin \mathcal{D}_r\}$ since free position on p in r are exactly those for which $n - o_r(pn)$ grows, and they are all enumerated.

We define again $\phi^{-1}_r(\{p_i n_i \Rightarrow a_i\}_{i=1}^k) = \{p_i \phi^{-1}_{r,p_i}(n_i) \Rightarrow a_i\}_{i=1}^k$, and it has the expected properties. □

Definition 4 (extraction) *Sub-stream extraction is defined as $r \uplus s = r \cdot \phi^{-1}_r(s)$, where r is the extracted sub-stream and $\phi^{-1}_r(s)$ is the rest after extraction.*

A consequence of the above proposition is that we can equivalently work only on "exploded" streams since they generate all streams. Indexes vary according to their ordering.

$$\begin{array}{lcll} \{pm \Rightarrow a\} \cdot \{qn \Rightarrow b\} & = & \{qn \Rightarrow b\} \cdot \{pm \Rightarrow a\} & p \neq q \\ \{pm \Rightarrow a\} \cdot \{pn \Rightarrow b\} & = & \{pn \Rightarrow b\} \cdot \{pm - 1 \Rightarrow a\} & m > n \end{array}$$

Examples Stream concatenation.

$$\begin{array}{l} \{2 \Rightarrow a\} \cdot (b,c) \cdot \{p \Rightarrow d\} \cdot \{q \Rightarrow e\} \cdot \{p \Rightarrow f\} \\ \quad = (b,a,c) \cdot \{p \Rightarrow d\} \cdot \{p \Rightarrow f\} \cdot \{q \Rightarrow e\} \\ \quad = \{\epsilon 1 \Rightarrow b, \epsilon 2 \Rightarrow a, \epsilon 3 \Rightarrow c, p1 \Rightarrow d, p2 \Rightarrow f, q1 \Rightarrow e\}. \end{array}$$

$$\begin{array}{l} \{p1 \Rightarrow a, p3 \Rightarrow b, r1 \Rightarrow c\} \cdot \{p1 \Rightarrow d, q_2 \Rightarrow e\} \\ \quad = \{p1 \Rightarrow a, p3 \Rightarrow b\} \cdot \{p1 \Rightarrow d\} \cdot \{q2 \Rightarrow e\} \cdot \{r1 \Rightarrow c\} \\ \quad = \{p1 \Rightarrow a, p2 \Rightarrow d, p3 \Rightarrow b, q2 \Rightarrow e, r1 \Rightarrow c\} \end{array}$$

4 Syntax of transformation calculus

In this section we define the untyped transformation calculus, and its simplified form the selective λ-calculus.

The definition is done in four steps. 1) We give a syntactic definition of terms in the transformation calculus, and add a structural equivalence on these terms. 2) Then we define substitutions on our terms, taking care of the constructs that did not appear in λ-calculus. 3) After that we need syntactical notions of *open transformation* and *juxtaposition*, to manipulate the structure of our terms. 4) Finally we define reduction rules for the transformation calculus.

$$
\begin{array}{lcll}
l & ::= & pn & p \in \mathcal{L}_s, n \in \mathcal{N} \\
M & ::= & x & \text{variable} \\
 & | & \{\} & \text{empty stream} \\
 & | & M \cdot \overline{\{l \Rightarrow x, \ldots\}} & \text{abstraction} \\
 & | & M \cdot \{l \Rightarrow M, \ldots\} & \text{application} \\
 & | & M \circ M & \text{composition}
\end{array}
$$

Figure 1: Syntax of the transformation calculus

$$
\begin{array}{lcll}
x[x\backslash N] & \equiv & N & \\
y[x\backslash N] & \equiv & y & x \neq y \\
\{\}[x\backslash N] & \equiv & \{\} & \\
(M \cdot \overline{R})[x\backslash N] & \equiv & (M \cdot \overline{R}) & x \in V(R) \\
(M \cdot \overline{R})[x\backslash N] & \equiv & (M[x\backslash N]) \cdot \overline{R} & x \notin V(R), V(R) \not\emptyset FV(N) \\
(M \cdot R)[x\backslash N] & \equiv & (M[x\backslash N]) \cdot (R[x\backslash N]) & \\
(M \circ M')[x\backslash N] & \equiv & (M[x\backslash N]) \circ (M'[x\backslash N]) &
\end{array}
$$

Figure 2: Definition of substitutions

Notation In the following definitions we will use the abbreviation $A \not\emptyset B$ for $A \cap B = \emptyset$.

Definition 5 *Terms of the transformation calculus, or Λ_T, are those generated by M in the grammar of figure 1, where variables should be distinct in abstractions, and labels distinct in streams. They are considered modulo $\equiv$, the minimal equivalence relation defined by the closure of the following two equalities:*

$$
\begin{array}{rl}
\forall R, S \in \mathcal{S}(\Lambda_T), & M \cdot R \cdot S \equiv M \cdot (R \cdot S) \\
\forall R, S \in \mathcal{S}(\mathcal{V}), V(R) \not\emptyset V(S) \Rightarrow & M \cdot \overline{R} \cdot \overline{S} \equiv M \cdot \overline{(S \cdot R)}
\end{array}
$$

where $V(S)$ is the set of variables of the anti-stream S.

The selective λ-calculus Λ_S is the subset formed by those terms which do not contain $\circ$ nor $\{\}$.

The new notation chosen for abstraction will be justified by its use in meta-operations on terms. We will of course accept all the abbreviations used in section 2 as long as there is no ambiguity. Composition binds weakly on its left and strongly on its right: $M \circ N \cdot \overline{\{l \Rightarrow x\}} \circ P \cdot \{l' \Rightarrow Q\}$ is equivalent to $(((M \circ N) \cdot \overline{\{l \Rightarrow x\}}) \circ P) \cdot \{l' \Rightarrow Q\}$.

Substitutions are done in the same way as for lambda-calculus, we only must pay attention to the new composition. Terms will always be considered modulo α-conversion. That is $M \cdot \overline{\{l \Rightarrow x\}} \equiv M[x\backslash y] \cdot \overline{\{l \Rightarrow y\}}$ when $y \notin FV(M)$.

Definition 6 Substitutions *are defined in figure 2, where $FV(M)$ is the set of free variables of M (resp. for streams of terms). We extend this definition to simultaneous substitution of multiple variables.*

Before defining composition we need an intermediate step, that of open transformations. Intuitively they are the syntactical counterpart for a sequence of operations we

$$
\begin{array}{lllll}
 & \text{Reduction} & & & \\
(\beta) & M \cdot \overline{\{l_i \Rightarrow x_i\}}_{i=1}^{n} \cdot \{l_i \Rightarrow N_i\}_{i=1}^{n} & \overset{n}{\rightarrow} & M[x_i \backslash N_i]_{i=1}^{n} & \\
(\circ) & M \circ (\{\} \odot P) & \rightarrow & M \odot P & BV(P) \not\sharp FV(M) \\
 & \text{Reordering} & & & \\
(\cdot\cdot) & M \cdot \overline{R} \cdot S & \rightarrow & M \cdot \phi_R^{-1}(S) \cdot \overline{\phi_S^{-1}(R)} & \mathcal{D}_R \not\sharp \mathcal{D}_S, V(R) \not\sharp FV(S) \\
(\cdot) & (M \circ N) \cdot R & \rightarrow & M \circ (N \cdot R) & \\
(\bar{\cdot}) & M \circ (N \cdot \overline{R}) & \rightarrow & (M \circ N) \cdot \overline{R} & FV(M) \not\sharp V(R)
\end{array}
$$

Figure 3: Reduction system for the transformation calculus.

do on a stream (to be read from right to left). However juxtaposition is not composition since it does not hide variable bounds by abstractions: an open transformation not only modifies the stream, but variable bindings too. To avoid that and keep variables local we will use (closed) transformations.

Definition 7 *An* open transformation *is a syntactic entity representing a list of streams of terms, anti-streams of variables, and terms. It can be linked to terms by the juxtaposition meta-operator "$\odot$".*

$$M \odot (e_1, \cdots, e_n) \stackrel{\text{def}}{=} (\ldots(M * e_1)\ldots) * e_n.$$

where $N * e = N \cdot e$ if e is a stream or an anti-stream, and $N * e = N \circ e$ if e is a term.

The set of variables bound by anti-streams in the body of an open transformation P is $BV(P)$.

This definition introduces a distinction between functional terms (which can be written $x \odot P$) and transformational terms (or closed transformations, since variable bound in their body are local, written $\{\} \odot P$). We call the atom appearing on the left of the juxtaposition operator *head* of the term.

Now we are ready to define rewriting rules for the transformation calculus.

Definition 8 *"$\rightarrow$" is defined on transformation calculus terms by the two reduction rules and three reordering rules of figure 3. "$\stackrel{*}{\rightarrow}$" is the combination in any order of $\rightarrow$ and $\equiv$.*

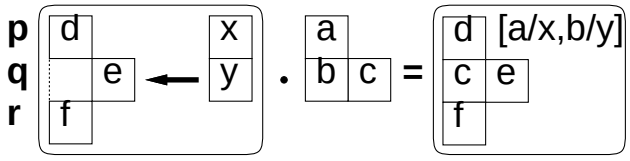
Selective λ-terms and rules $(\beta, \cdot\cdot)$ define the selective λ-calculus.

One may think of many other axiomatizations of this calculus, keeping the same intuitive semantics. This one focuses on both β-reduction and composition. Other rules are conceived to give the maximal freedom to reductions, while preserving confluence, and the existence of normal forms. Figure 4 shows how reduction works on normal forms.

Rule β is the classical β-reduction extended to streams. An index n is added to keep meaningful the number of β-steps along a reduction path.

Rule $\circ$ "connects" a function and a transformation. The condition, that all variables bound in P should be protected in M, means that we "close" our transformation, hiding its variables from M, before syntactically juxtaposing it.

- Application



- Composition

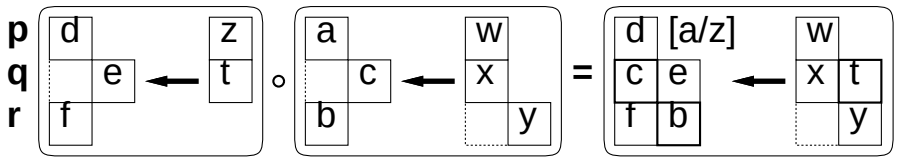


Figure 4: Transformation application and composition

Reordering rules "$(\cdot)$" and "$(\bar{\cdot})$" are easily understandable. They respectively send the input of a composition to its right-hand side, and put outside abstractions of its right-hand side. Only one of them would be enough, but we give the two for the sake of symmetry.

Rule $(\cdot\cdot)$ is the symmetrical of β-reduction, and is based on the equality $N \cdot \overline{(R \uplus S)} \cdot (R' \uplus S') \equiv N \cdot \overline{\phi_S^{-1}(R)} \cdot \overline{S} \cdot R' \cdot \phi_R^{-1}(S')$ (with $\mathcal{D}_{R'} = \mathcal{D}_R, \mathcal{D}_{S'} = \mathcal{D}_S$ and $V(S) \not\subset V(R)$). If we take $M = N \cdot \overline{\phi_S^{-1}(R)}$, and apply $M \cdot \overline{S} \cdot R'$ to $\phi_R^{-1}(S')$, then rule $(\cdot\cdot)$ preserves confluence: it gives $N \cdot \overline{(R \uplus S)} \cdot (R' \uplus S') \to N \cdot \overline{\phi_S^{-1}(R)} \cdot \phi_S^{-1}(R') \cdot \overline{\phi_R^{-1}(S)} \cdot \phi_R^{-1}(S')$.

Theorem 1 *Selective λ-calculus is confluent.*

Theorem 2 *Transformation calculus is confluent.*

The proofs are given in the next section. Confluence of transformation calculus is obtained from selective λ-calculus through a translation into it.

Proposition 3 *Normal terms are generated by N in the following grammar, where streams and anti-streams may be empty.*

$$\begin{array}{lcl} H & ::= & x \mid \{\} \\ F & ::= & x \cdot \{l \Rightarrow N, \ldots\} \mid F \cdot \overline{\{l \Rightarrow x, \ldots\}} \circ F \\ N & ::= & H \cdot \{l \Rightarrow N, \ldots\} \cdot \overline{\{l \Rightarrow x, \ldots\}} \mid N \circ F \cdot \overline{\{l \Rightarrow x, \ldots\}} \end{array}$$

The definition of F in the above proposition means that, in a normal form, only a functional term may appear on the right side of a composition.

5 Applicative translation and confluence

The idea of applicative continuation semantics was developed in [10] for a framework of stores and partial continuations. A partial continuation is a function from a state to a new state, but we can translate it into a function prefixing a continuation, that is

a function from a continuation to a new continuation doing intended operations before calling the old continuation. In fact transformations look very much like partial continuations on streams, and this idea provides us with a translation from transformation calculus to selective λ-calculus.

Definition 9 *Tr is the applicative translation from $\Lambda_T(\mathcal{L}_s)$ to $\Lambda_S(\mathcal{L}_s \cup \{cont\})$. It is extended to streams by structural mapping.*

$$\begin{array}{rcl} Tr(\{\}) & = & x \cdot \overline{\{cont \Rightarrow x\}} \\ Tr(M \circ N) & = & Tr(N) \cdot \{cont \Rightarrow Tr(M)\} \\ Tr(x) & = & x \\ Tr(M \cdot \overline{R}) & = & Tr(M) \cdot \overline{R} \\ Tr(M \cdot R) & = & Tr(M) \cdot Tr(R) \end{array}$$

The image of Λ_T is $Tr(\Lambda_T) = \Lambda_S^*$.

"*cont*" in the above definition stands for continuation. We translate composition into an application of its right-hand side to its left-hand side, seen as a continuation. Since *cont* is a new name, the continuation is received by the $x \cdot \overline{\{cont \Rightarrow x\}}$ head of the right-hand side, and semantics of the calculus are preserved.

We will use this translation to prove the confluence of transformation calculus, based on that of selective λ-calculus. We do not give here detailed proofs, but just point out what makes them work.

Theorem 1 *Selective λ-calculus is confluent.*

$$\forall M,P,Q\ (M \xrightarrow{*} P \wedge M \xrightarrow{*} Q) \Rightarrow (\exists T\ P \xrightarrow{*} T \wedge Q \xrightarrow{*} T)$$

PROOF We can adapt from [2] that selective λ-calculus in the definition we give here is confluent. We have to modify two points. First, in its original definition selective λ-calculus use sum labels ($\mathcal{L} = \mathcal{N} \cup \mathcal{L}_s$) whereas we use a product system here ($\mathcal{L} = \mathcal{L}_s \times \mathcal{N}$). Names and indexes being independent, the modification of the proof is immediate. The modified proof appears in [6]. Next, structural equivalences were defined in terms of reordering rules included in the calculus. Once proved the confluence of the calculus including them, we can transform these rules into equivalences and keep confluence. □

Lemma 1 *If $M \xrightarrow{*} N$ and $M,N \in \Lambda_S^*$ then there is a reduction path $M \to M_1 \to \ldots \to N, M_i \in \Lambda_S^*$. Moreover selective λ-calculus restricted to Λ_S^* is confluent.*

PROOF (sketch) For the first part, this is similar to a cut-elimination lemma, and we prove it by induction on the reduction path. The two steps that get a term out of Λ_S^* are $x \cdot \overline{\{cont \Rightarrow x\}} \cdot \{l \Rightarrow M\} \to x \cdot \{l \Rightarrow M\} \cdot \overline{\{cont \Rightarrow x\}}$ and $M \cdot \overline{\{l \Rightarrow x\}} \cdot \{cont1 \Rightarrow P, cont2 \Rightarrow Q\} \to M \cdot \{cont2 \Rightarrow Q\} \cdot \overline{\{l \Rightarrow x\}} \cdot \{cont1 \Rightarrow P\}$, and they are suppressible since anyway one cannot use the head x without reducing the abstraction on $cont1$, and there will never be an abstraction on $cont2$.

For the second part, if our confluence term is out of Λ_S^* then we can eliminate the steps getting it out of Λ_S^* on the two sides for the same reasons, and the two reductions will still end on an identical term, in Λ_S^* this time. □

We were speaking above of "keeping semantics" without defining what it means. Here is a possible definition, which does not only mean that two systems have same models, but sets a relation between their reductions too.

Definition 10 *We will say that two reduction systems $(R_1, \rightarrow_1)$ and $(R_2, \rightarrow_2)$ are mutually confluent for some couple of translations $(Tr_1 : R_1 \rightarrow R_2, Tr_2 : R_2 \rightarrow R_1)$, such that $\forall M \in R_1\ M \xrightarrow{*}_1 Tr_2(Tr_1(M))$ and converse, whenever the following property and its converse (exchanging 1 and 2) are verified:*

$$\forall M, N, P \in R_1$$
$$(M \xrightarrow{*}_1 P \wedge N \xrightarrow{*}_1 P) \Rightarrow (\exists T \in R_2)(Tr_1(M) \xrightarrow{*}_2 T \wedge Tr_1(N) \xrightarrow{*}_2 T)$$

Lemma 2 *When two reduction systems are mutually confluent their confluence problems are equivalent.*

PROOF Suppose $(R_2, \rightarrow_2)$ is confluent. Take M, P, Q in R_1, $M \xrightarrow{*}_1 P$ and $M \xrightarrow{*}_1 Q$. Then by mutual confluence on these reductions combined with respectively $P \xrightarrow{*}_1 P$ and $Q \xrightarrow{*}_1 Q$, we have P', Q' such that $Tr_1(M) \rightarrow P', Tr_1(M) \xrightarrow{*}_2 Q', Tr_1(P) \xrightarrow{*}_2 P', Tr_1(Q) \xrightarrow{*}_2 Q'$. By confluence of $\rightarrow_2$ we get T such that $P' \xrightarrow{*}_2 T$ and $Q' \xrightarrow{*}_2 T$. Using mutual confluence back on $Tr_1(P) \xrightarrow{*}_2 T$ and $Tr_1(Q) \xrightarrow{*}_2 T$ we get T' such that $Tr_2(Tr_1(P)) \xrightarrow{*}_1 T'$ and $Tr_2(Tr_1(Q)) \xrightarrow{*}_2 T'$. By the condition on $Tr_2 \circ Tr_1$ we have $P \xrightarrow{*}_1 T'$ and $Q \xrightarrow{*}_1 T'$. □

Lemma 3 *$(\Lambda_T, \rightarrow)$ and $(\Lambda_S^*, \rightarrow)$ are mutually confluent.*

PROOF For Tr_1 we take Tr, but for Tr_2 we cannot simply take Tr^{-1} since $(\cdot)$ is mapped on a structural equivalence. We choose $Tr_2(M)$ as being the $(\cdot)$-normal form of antecedents of M by Tr (it is itself an antecedent of M). With that we have $(\forall M \in R_1) M \xrightarrow{*} Tr_2(Tr_1(M))$ and $Tr_1 \circ Tr_2 = Id$.

We have $\forall M, N \in \Lambda_T\ (M \xrightarrow{*} N) \Rightarrow (Tr(M) \xrightarrow{*} Tr(N))$, ($\beta$ and $(\cdot\cdot)$ are the same, $(\circ)$ is translated into n steps of $(\cdot\cdot)$ and 1 of β, $(\cdot)$ disappears and $(\bar{\cdot})$ becomes $(\cdot\cdot)$), and the first direction as a consequence.

The second direction is more complex, since we may apply structural equivalences in Λ_S^* that are not valid in Λ_T. This happens when we apply $(\cdot\cdot)$ in presence of *cont*: $M \cdot \overline{\{l \Rightarrow x\}} \cdot \{cont \Rightarrow P\} \cdot \{l_1 \Rightarrow N_1, \ldots\}$ will be translated into $Tr_2(M) \cdot \overline{\{l \Rightarrow x\}} \cdot \{l_1 \Rightarrow Tr_2(N_1), \ldots\} \circ Tr_2(P)$, so we should apply $(\cdot\cdot)$ only on the $M \cdot \overline{\{l \Rightarrow x\}} \cdot \{l_1 \Rightarrow N_1, \ldots\} \cdot \{cont \Rightarrow P\}$ form, which may force us to use β before $(\cdot\cdot)$ if l is one of l_i's. If we apply such a transformation on the two branches $M \xrightarrow{*} P$ and $N \xrightarrow{*} P$, and equalize them by doing in each branch the supplementary β-reductions or reorderings that were done in the other, we obtain reductions directly translatable into the transformation calculus. □

Theorem 2 *Transformation calculus is confluent.*

PROOF Corollary of lemma 3. □

6 Scope-free variable encoding

We cannot expect to give a precise definition of *scope-free variable* in the transformation calculus. It appears as an intuitive notion of a variable whose locality is not syntactical but operational. Since transformation calculus has no operational semantics, definition seems impossible. We will do it outside of the calculus.

Definition 11 *A* scope-free variable *is some way to create, modify and destroy a value such that:*

1. *these operations may appear in different syntactic entities, which may be used independently.*
2. *a correct use of this variable is obtained when a creator, some modifiers, and a destructor result in a* modification sequence *(that is, modifications must be ordered).*
3. *its correct use in a modification sequence has no side-effect outside it.*

As we introduced in section 2, elementary creators, modifiers and destructors in transformation calculus are respectively $\{\} \cdot \{v \Rightarrow M\}$, $\{\} \cdot \{v \Rightarrow M\} \cdot \overline{\{v \Rightarrow x\}}$ and $\{\} \cdot \overline{\{v \Rightarrow x\}}$. But we can think of more complex ones, acting simultaneously on multiple variables, taking arguments, or returning results. For instance, in

$$((x+y) \cdot \overline{\{a \Rightarrow x, b \Rightarrow y\}}) \circ (\{a \Rightarrow x \times y, b \Rightarrow x - y\} \cdot \overline{\{a \Rightarrow x, b \Rightarrow y\}}) \circ (\{a \Rightarrow 2, b \Rightarrow x\} \cdot \overline{\{1 \Rightarrow x\}})$$

a and b are two scope-free variables, but their creator, modifier and destructor are joint.

To ensure that we have a correct scope-free variable encoding here, we must verify the third point of the definition, which says that it has no effect outside the sequence it is used in.

Proposition 4 *The scope-free variable encoding into the transformation calculus ensures locality to the modification sequence.*

PROOF If our variable is encoded on a label whose name is unique, there is no problem since it does not modify indexes of other arguments and variables, and its label appears only between its creation and destruction.

Otherwise, labels with the same name may appear with a different index in the sequence. But since each time they meet an abstraction or an application on our variable their indexes go alternately up and down, and the number of abstraction is equal to the number of applications, their index is the same on each side of the sequence. Moreover modifiers do not modify the value they support, since index changes avoid overloading of the same label. □

We claim that, thanks to this property, scope-free variables are not only more flexible than classical scoped variables, but can replace them in most of their uses. Particularly, in a functional language, they can replace references without the need of a strict evaluation scheme. The only condition is that each scope-free variable that may be used in the same sequence should be given a label on a different name.

7 Conclusion

We proposed the transformation calculus as an extension of lambda-calculus permitting both functional and imperative encoding of algorithms.

We think this is a satisfactory solution to the two needs exposed in introduction: as a demonstration of the relation between lambda-calculus and algorithms, and as a basis for functional languages handling states and sequentiality problems.

Still there are many topics left to explore. Typing is one of them. If we are to write program in this calculus, it is even unavoidable, since we must be able to verify that scope-free variables are correctly used. The polymorphically typed version of selective λ-calculus presented in [7], extended to the transformation calculus, would be an interesting framework.

Compilation, which is easy with stores, is complex here. The final goal would be to eliminate label information, but the possibility to compose a term with a variable makes impossible to reach it in the general case. There is no problem if we use non-polymorphic typing, but the polymorphic case is still open.

One strength of the transformation calculus is its system of labels. We may be interested in extending it. For instance, the possibility to generate new label names would give unique identifiers for scope-free variable, and avoid the hiding of a label in those sub-sequences which create new variables on this label. A more structured label space could even enable the use of object-based techniques.

Last, the similarities between this calculus and process calculi suggest that it might be used to express some forms of parallelism. If one looks at the way data flows in our terms, lots of reminiscences of the dataflow model may be seen. A topic like compilation of the calculus into this model looks interesting.

References

[1] Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, October 1991.

[2] Hassan Aït-Kaci and Jacques Garrigue. Label-selective λ-calculus: Syntax and confluence. In *Proc. of the conf. on Foundations of Software Technology and Theoretical Computer Science*, pages 24–40. Springer-Verlag LNCS 761, 1993.

[3] Gérard Boudol. Towards a lambda-calculus for concurrent and communicating systems. In *Proceedings of TAPSOFT '89*, pages 149–161, Berlin, Germany, 1989. Springer-Verlag, LNCS 351.

[4] Guy Cousineau, Pierre-Louis Curien, and Michel Mauny. The categorical abstract machine. *Science of Computer Programming*, 8, 1987.

[5] Pierre-Louis Curien. *Categorical Combinators, Sequential Algorithms and Functional Programming.* Research Notes in Theoretical Computer Science. Pitman, 1986.

[6] Jacques Garrigue. Lambda-calcul à sélection par labels et calcul des transformations. Mémoire de magistère MMFAI, École Normale Supérieure, Paris, 1993.

[7] Jacques Garrigue and Hassan Aït-Kaci. The typed polymorphic label-selective λ-calculus. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 35–47, 1994.

[8] David K. Gifford and John M. Lucassen. Integrating functional and imperative programming. In *Proc. ACM Conference on LISP and Functional Programming*, pages 28–38, 1986.

[9] Thérèse Hardin. Confluence results for the pure strong categorical logic CCL. λ-calculi as subsystems of CCL. *Theoretical Computer Science*, 65:291–342, 1989.

[10] Gregory F. Johnson and Dominic Duggan. Stores and partial continuations as first-class objects in a language and its environment. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 158–168, 1988.

[11] John Lamping. A unified system of parameterization for programming languages. In *Proc. ACM Conference on LISP and Functional Programming*, pages 316–326, 1988.

[12] Robin Milner. The polyadic π-calculus: A tutorial. LFCS Report ECS-LFCS-91-180, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, October 1991.

[13] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1991.

[14] Martin Odersky, Dan Rabin, and Paul Hudak. Call by name, assignment, and the lambda calculus. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 43–56, 1993.

[15] Simon L. Peyton Jones and Philip Wadler. Imperative functional programming. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 71–84, 1993.

[16] Jean-Pierre Talpin and Pierre Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, 2(3):245–271, July 1992.

[17] Philip Wadler. Comprehending monads. In *Proc. ACM Conference on LISP and Functional Programming*, pages 61–78, 1990.